SOEN2070 C++程序设计

07 迭代器

刘安 anliu@suda.edu.cn 2023-2024-2

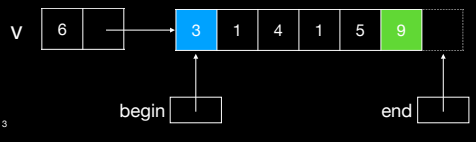
再访迭代器

迭代器和序列紧密相关

- 数组和vector都可以看成一个序列，每个序列都有一个头和一个尾
- 迭代器就是指向序列中某个元素的对象
- begin迭代器：指向序列的头
- end迭代器：指向序列的尾后元素

```
int a[6] = {3, 1, 4, 1, 5, 9};
```

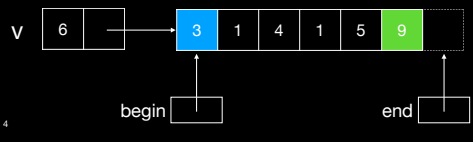

```
std::vector<int> v {3, 1, 4, 1, 5, 9};
```



迭代器支持的基本操作

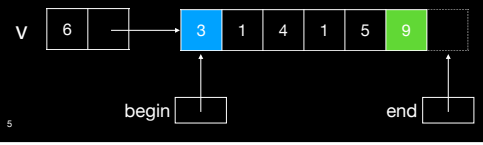
- 迭代器指向序列中的某个元素或者尾后元素
- 可以通过==和!=比较指向同一序列的两个迭代器
  - p==q当且仅当p和q指向同一个元素或者均指向尾后元素
  - p!=q当且仅当!(p==q)
- 可以通过++p让p指向当前元素的下一元素
- 可以通过*p获取p当前指向元素的引用

```
std::vector<int> v {3, 1, 4, 1, 5, 9};
```



迭代器的经典用法

```
std::vector<int> v {3, 1, 4, 1, 5, 9};
for (auto iter { v.begin() }; iter != v.end(); ++iter)
    std::cout << *iter << '\n';
```

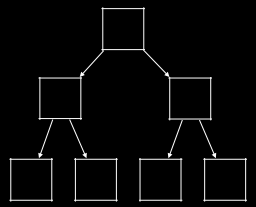


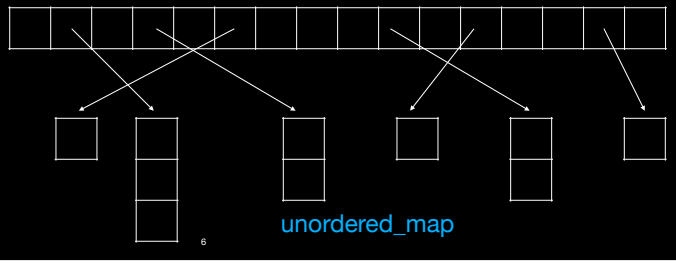
为什么使用迭代器来遍历容器

- 使用下标运算不可以吗?

```
for (int i { 0 }; i < v.size(); ++i)
    std::cout << v[i] << '\n';
```

- 对于关联容器，比如map和unordered_map，如何使用下标运算?





为什么使用迭代器来遍历容器

- 迭代器可以在任何容器上进行遍历，无论底层数据是如何存储的

```
int count(const std::vector<int>& c, int v)
{
    int cnt { 0 };
    for (auto iter { c.begin() }; iter != c.end(); ++iter)
        if (*iter == v)
            ++cnt;
    return cnt;
}
```

上述代码可以用于std::list吗?

为什么使用迭代器来遍历容器

- 迭代器可以在任何容器上进行遍历，无论底层数据是如何存储的

```
int count(const std::vector<int>& c, int v)
{
    int cnt { 0 };
    for (auto iter { c.begin() }; iter != c.end(); ++iter)
        if (*iter == v)
            ++cnt;
    return cnt;
}
```

上述代码可以用于std::list吗?

为什么使用迭代器来遍历容器

- 迭代器可以在任何容器上进行遍历，无论底层数据是如何存储的

```
int count(const std::list<int>& c, int v)
{
    int cnt { 0 };
    for (auto iter { c.begin() }; iter != c.end(); ++iter)
        if (*iter == v)
            ++cnt;
    return cnt;
}
```

上述代码可以用于std::set吗?

为什么使用迭代器来遍历容器

- 迭代器可以在任何容器上进行遍历，无论底层数据是如何存储的

```
int count(const std::list<int>& c, int v)
{
    int cnt { 0 };
    for (auto iter { c.begin() }; iter != c.end(); ++iter)
        if (*iter == v)
            ++cnt;
    return cnt;
}
```

上述代码可以用于std::set吗?

为什么使用迭代器来遍历容器

- 迭代器可以在任何容器上进行遍历，无论底层数据是如何存储的

```
int count(const std::set<int>& c, int v)
{
    int cnt { 0 };
    for (auto iter { c.begin() }; iter != c.end(); ++iter)
        if (*iter == v)
            ++cnt;
    return cnt;
}
```


std::map<Key, Value>的迭代器

std::map<Key, Value>的迭代器

- 迭代器指向std::pair<Key, Value>
  - 类模板std::pair<T1, T2>在头文件<utility>中定义
  - 将两个可能属于不同类型的值组合起来
  - 通过first和second成员访问这两个值

```
std::pair<std::string, int> p;
p.first = "id";
p.second = 1001;
```


std::map<Key, Value>的迭代器

- 迭代器指向std::pair<Key, Value>
  - 类模板std::pair<T1, T2>在头文件<utility>中定义
  - 将两个可能属于不同类型的值组合起来
  - 通过first和second成员访问这两个值

```
std::map<std::string, int> m;
for (auto iter { m.begin( )}; iter != m.end(); ++iter)
    std::cout << (*iter).first << ' ' << (*iter).second << '\n';
    //std::cout << iter->first << ' ' << iter->second << '\n';
```


std::pair<T1, T2>

- 构造pair的两种方式

```
std::pair<std::string, int> p { "id", 1001 };
auto q { std::make_pair("id", 1001) };
```

- make_pair可以自动推断出类型，所以可以使用auto
- 第一种方式不能使用auto

迭代器类别

迭代器类别

- 目前对迭代器的主要操作是++iter
- 然而，有时候我们希望如下操作

```
std::vector<int> v(10);
auto mid = v.begin() + v.size() / 2;

std::deque<int> d(13);
auto some_iter = d.begin() + 3;
```

迭代器类别

- 下面代码在std::list上表现如何

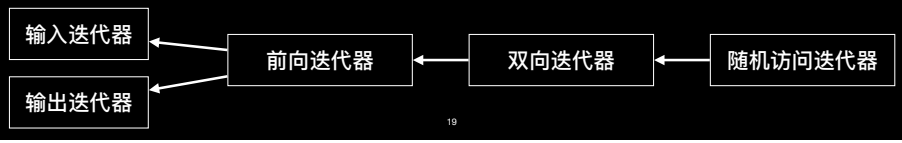
```
std::list<int> my_list(10);
auto some_iter = my_list.begin() + 3;
```

Invalid operands to binary expression ('iterator' (aka '__list_iterator<int, void *>') and 'int')

为什么list上的迭代器不支持如上操作?

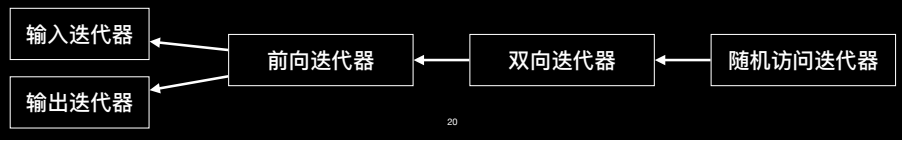
迭代器有5种类别

- 输入迭代器 Input
- 输出迭代器 Output
- 前向迭代器 Forward
- 双向迭代器 Bidirectional
- 随机访问迭代器 Random access



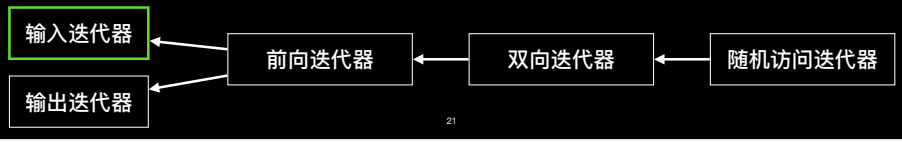
所有迭代器都支持的操作

- 拷贝 `auto iter = v.begin();`
- 通过++向前移动 `++iter;`
- 通过==和!=进行比较 `if (iter != v.end())`



输入迭代器

- 只读，解引用只能出现在表达式右侧　`int val = *iter;`
- 单遍扫描 single-pass
  - 从数据流中读取数据，读完之后该数据可能不再存在，因此无法重复读取



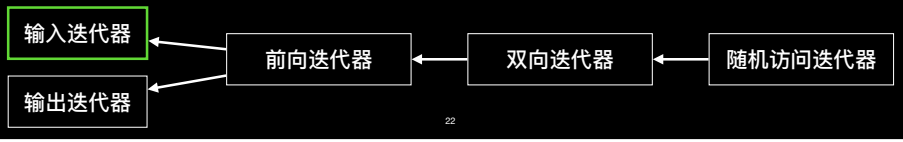
输入迭代器

- 只读，解引用只能出现在表达式右侧　`int val = *iter;`
- 单遍扫描 single-pass
- 使用场景
  - std::find
  - 输入流

std::find, std::find_if, std::find_if_not

Defined in header <algorithm>

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```



输出迭代器

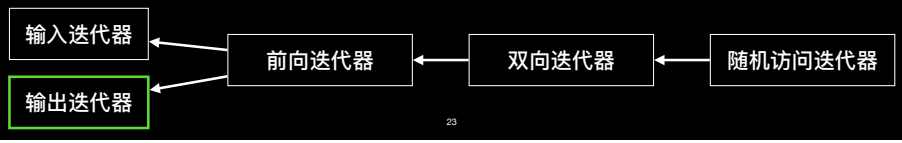
- 只写，解引用只能出现在表达式左侧　`*iter = 42;`
- 单遍扫描 single-pass
- 使用场景
  - std::copy
  - 输出流

C++ Algorithm library

std::copy, std::copy_if

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last,
               OutputIt d_first );
```



前向迭代器

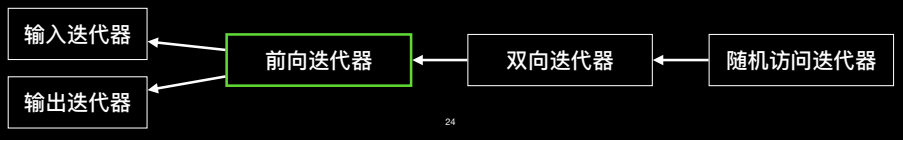
- 可读，可写（元素是非const）
- 多遍扫描 multiple-passes
- 使用场景
  - std::replace
  - std::forward_list

C++ Algorithm library

std::replace, std::replace_if

Defined in header <algorithm>

```
template< class ForwardIt, class T >
void replace( ForwardIt first, ForwardIt last,
              const T& old_value, const T& new_value );
```



双向迭代器

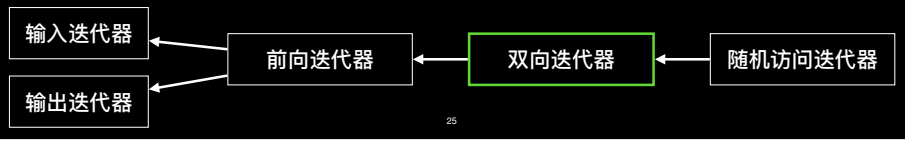
- 在前向迭代器基础上，支持使用--向后移动
- 使用场景
  - std::reverse
  - std::list, std::map, std::set

C++ | Algorithm library

std::reverse

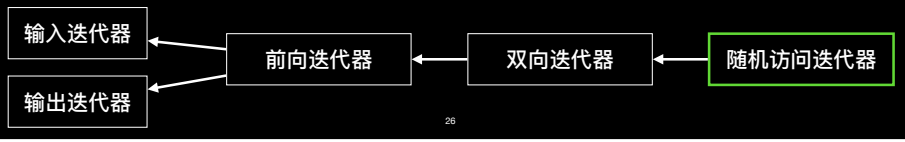
Defined in header <algorithm>

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```



随机访问迭代器

- 在双向迭代器基础上，支持使用+和-向前或向后移动多步
  - 指向同一序列的两个迭代器相减得到它们之间的距离
- 使用场景
  - std::vector, std::deque, std::string
  - 指针



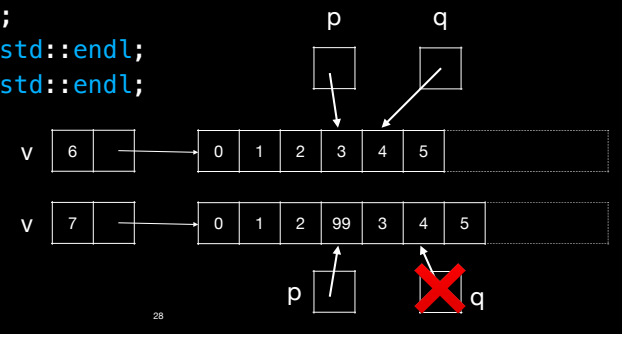
迭代器陷阱


```
std::vector<int> v { 0, 1, 2, 3, 4, 5 };
auto p { v.begin() };
p += 3; //随机访问迭代器vector<T>::iterator
auto q { p };
++q;
p = v.insert(p, 99);
std::cout << *p << std::endl;
std::cout << *q << std::endl;
```

输出结果

99

0



迭代器陷阱

- 随着vector的大小增长，可能会为其元素分配新的内存
- 这样迭代器q指向的空间可能会变成无效空间

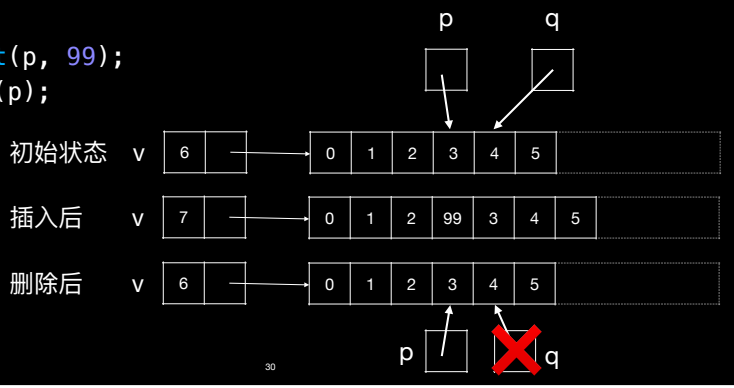
```
std::vector<int> v { 0, 1, 2, 3, 4, 5 };
auto p { v.begin() };
p += 3; //随机访问迭代器vector<T>::iterator
auto q { p };
++q;
std::cout << v.size() << ' ' << v.capacity() << std::endl;
p = v.insert(p, 99);
std::cout << v.size() << ' ' << v.capacity() << std::endl;
```

输出结果

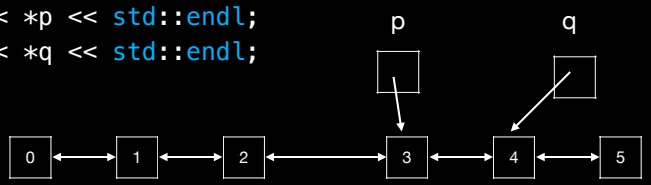
6 6

7 12


```
std::vector<int> v { 0, 1, 2, 3, 4, 5 };
auto p { v.begin() };
p += 3; //随机访问迭代器vector<T>::iterator
auto q { p };
++q;
p = v.insert(p, 99);
p = v.erase(p);
```

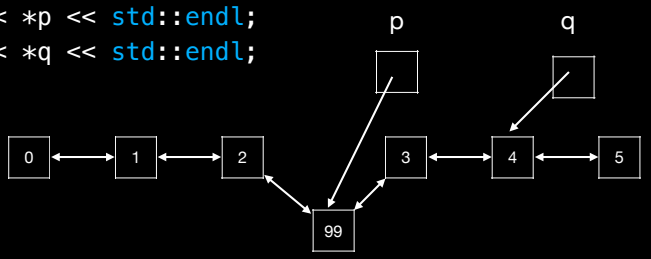



```
std::list<int> v { 0, 1, 2, 3, 4, 5 };
auto p { v.begin() };
++p; ++p; ++p; //双向迭代器list<T>::iterator
auto q { p };
++q;
p = v.insert(p, 99);
std::cout << *p << std::endl;
std::cout << *q << std::endl;
```




```
std::list<int> v { 0, 1, 2, 3, 4, 5 };
auto p { v.begin() };
++p; ++p; ++p; //双向迭代器list<T>::iterator
auto q { p };
++q;
p = v.insert(p, 99);
std::cout << *p << std::endl;
std::cout << *q << std::endl;
```

p和q始终有效



迭代器失效规则 - 删除元素

- 指向被删元素的迭代器始终失效
- vector/string：被删元素之后的所有迭代器均失效
  - 被删元素之前的所有迭代器均有效
- deque：被删元素如果不是首尾元素，那么所有迭代器均失效
  - 被删元素是尾元素，那么尾后迭代器失效
- list/forward_list/set/map：所有其他迭代器均有效

迭代器失效规则 - 插入元素

- vector/string：被插元素之后的所有迭代器均失效
  - 如果存储空间重新分配，所有迭代器均失效
- deque：所有迭代器均失效
- list/forward_list/set/map：所有迭代器均有效

一段有问题的代码

```
void erase_all_buggy(vector<int>& v, int val)
{
    for (auto iter { v.begin() }; iter != v.end(); ++iter) {
        if (*iter == val)
            v.erase(iter);
    }
}

std::vector<int> v { 3, 1, 1, 4, 1, 5 };
erase_all_buggy(v, 1);
```

使用失效的迭代器

迭代器在erase之后失效

输出结果

3 1 4 5

不要使用失效迭代器

```
void erase_all(vector<int>& v, int val)
{
    for (auto iter { v.begin() }; iter != v.end(); ) {
        if (*iter == val)
            iter = v.erase(iter);
        else
            ++iter;
    }
}
```

这里不修改迭代器

erase返回一个有效的迭代器
指向被删元素之后的元素

如果没有删除元素
迭代器有效，可以使用